Change Distilling: Software Evolutionsanalyse mit Hilfe von feingranularen Programmtext-Änderungen

Beat Fluri

s.e.a.l. – software evolution and architecture lab
Institut für Informatik der Universität Zürich, Schweiz
fluri@ifi.uzh.ch

Abstract: Da sich bestehende Software ständig verändert, wird sie grösser, komplizierter und daher weniger wartbar. Um dieser Degeneration entgegen zu wirken, befasst sich die Forschung im Bereich Software Engineering unter anderem mit der Evolutionsanalyse bestehender Software Systeme. Die zu Grunde liegende Idee ist, dass wir anhand der Geschichte eines Software Systems lernen können, wie es zur Degeneration gekommen ist und wie wir diese gewonnene Information einsetzten können, um die Software-Entwicklungsarbeit künftig zu unterstützen. In der hier zusammengefassten Dissertation wird die Methodik *Change Distilling* vorgestellt. Sie erlaubt es wichtige Informationen über den Softwareänderungsprozess zu extrahieren, zu verstehen und zur Unterstützung der Entwicklung von Software einzusetzen. Diese Methodik beschreibt, wie feingranulare Programmtext-Änderungen klassifiziert und extrahiert werden können. Anhand drei empirischen Studien wird gezeigt, wie wir das generierte Wissen über die Geschichte der einzelnen Software Systeme zu deren Verbesserung einsetzen können.

1 Einleitung

Bereits anfangs der Achtziger Jahre hat Lehmann postuliert, dass Software ständige Anpassungen erfahren muss, um weiterhin von Nutzen zu sein [Leh80]. Die Gründe warum Software sich kontinuierlich ändern muss sind mannigfaltig: Funktionalität wird ergänzt; Fehler müssen behoben werden; oder die Software muss wegen Modernisierung der Infrastruktur einem Reengineering unterzogen werden. Ein negativer Effekt dieser kontinuierlichen Änderung ist das Phänomen der Software Alterung. Da Software von Leuten geändert wird, die von derem anfänglichem Entwurf keine Kenntnis haben und meistens unter Zeitdruck arbeiten müssen, vergrössert sich die Software, wird komplizierter und weniger verständlich. Aus diesem Grund wurden im letzten Jahrzehnt verschiedene Techniken entwickelt, um ein besseres Verständnis der negativen Auswirkungen der kontinuierlichen Anpassung zu erlangen. Diese Techniken analysieren Änderungen im Allgemeinen und Programmtext-Änderungen im Besonderen.

Bis anhin entwickelte Ansätze weisen starke Einschränkungen auf, weil sie auf grobgranularen Änderungsinformationen aus Versionierungssystemen (z. B. *CVS*, *Subversion*, *ClearCase*, etc.) zurückgreifen. Diese führen über Dateiänderungen Buch, indem sie ledig-

lich textuelle Unterschiede speichern. Damit ist es jedoch nicht möglich die syntaktische Natur der Änderungen der Programmentitäten (z. B. Methoden, if-Anweisungen, etc.) zu erfassen. Zudem fehlen sowohl eine klare Definition von Programmtext-Änderungsarten als auch deren Klassifizierung. Um letztendlich die negativen Auswirkungen der kontinuierlichen Änderung zu verstehen, sind eine solche Definition und Klassifizierung zur Extraktion und Analyse von Programmtext-Änderungen äusserst wichtig. In der hier zusammengefassten Dissertation wird deshalb folgende Forschungsthese aufgestellt [Flu08]: *Die Extraktion, Klassifikation und Analyse von feingranularen Programmtext-Änderungen aus der Geschichte eines Software Systems liefern nützliche Erkenntnisse über die Probleme, welche durch die kontinuierlichen Änderung entstehen, und führen zu Techniken, um diese zu vermindern.*

Der Hauptbeitrag der Dissertation ist *Change Distilling*, eine Methodik zur Klassifikation, Extraktion und Analyse von feingranularen Programmtext-Änderungen (Abschnitt 2). *Change Distilling* liefert eine Taxonomie, welche Programmtext-Änderungsarten, anhand von Baum-Transformationen im abstrakten Syntax-Baum (AST) definieren. Um solche Transformationen zu extrahieren, wendet der *Change-Distilling*-Algorithmus Baum-Vergleiche auf aufeinanderfolgende Versionen eines ASTs an. Der Algorithmus wurde als Erweiterung in die Entwicklungsumgebung (IDE) *Eclipse*[1] für die Java Programmiersprache integriert und trägt den Namen *ChangeDistiller*.

Um den Nutzen der Extraktion feingranularer Änderungsarten zu zeigen und somit die Forschungsthese zu bestätigen, sind in Rahmen der Dissertation drei empirische Studien durchgeführt worden.

Erstens wurde in der Geschichte von sechs Software Systemen das Verhalten von gemeinsamen Änderungen zwischen Programmtext und Programmkommentaren analysiert (Abschnitt 3).

Zweitens wurde erkundet, ob gewisse Änderungsarten oft miteinander auftreten. Die Anwendung von hierarchischem agglomerativem *Clustering* an drei Software Systemen liessen Muster unter Änderungsarten erkennen (Abschnitt 4).

Drittens wurde untersucht, ob Methoden existieren, deren Aufrufe von Kontextänderungen (z. B. Verschiebung in eine if-Anweisung) signifikant stärker betroffen sind als andere und ob Muster zwischen diesen Aufruf-Änderungen zu finden sind (Abschnitt 5).

Diese Studien haben gezeigt, dass mit der *Change Distilling* Methodik die Geschichte eines Software Systems besser zu verstehen ist und dass diese Erkenntnisse die Entwicklung von Software vereinfachen können.

Dieser Artikel fasst die *Change Distilling* Methodik und die Resultate der drei Studien zusammen. Des Weiteren zeigt er anhand typischer Anwendungsszenarien Wege auf, wie Software Entwickler während ihrer tagtäglichen Arbeit mit Hilfe von Werkzeugen unterstützt werden können.

[1] http://www.eclipse.org

2 Extraktion von Programmtext-Änderungen

Programmtext wird als abstrakter Syntax-Baum (AST) repräsentiert. Der *Change-Distilling*-Algorithmus vergleicht die ASTs von zwei aufeinander folgende Versionen einer Klasse [FWPG07]. Der Algorithmus berechnet eine Braum-Transformations-Sequenz, die den älteren AST in den neueren AST transformieren. Dazu verwendet er die elementaren Transformationen Einfügen (*insert*), Löschen (*delete*), Verschieben (*move*) und Aktualisieren (*update*) eines AST-Knotens.

Die *Change-Distilling*-Taxonomie von Programmtext-Änderungen definiert Änderungsarten aufgrund von atomaren Baum-Transformationen im AST. Anhand der Taxonomie übersetzt der Algorithmus die generierte Transformations-Sequenz in konkrete Programmtext-Änderungen. Zusätzlich definiert die Taxonomie die Signifikanz von Änderungen. Diese so genannte Änderungssignifikanz (*change significance level*) drückt sowohl die mögliche Auswirkung einer Änderung auf andere Programmentitäten aus, als auch, ob die Änderung die Funktionalität des Programms verändern kann.

Zur Zeit deckt die Taxonomie über 40 verschiedene Änderungsarten ab [FG06], welche in Rumpf- (*body-part*) und Deklarations-Änderungen (*declaration-part*) unterschieden werden. Rumpf-Änderungen beziehen sich auf Änderungen innerhalb von Klassen und Methoden, Deklarations-Änderungen auf Deklarationen von Attributen, Klassen und Methoden. Jeder Änderungsart wird eine Änderungssignifikanz von *keine* (*none*), *tief* (*low*), *mittel* (*medium*), *hoch* (*high*) oder *kritisch* (*crucial*) zugewiesen. Für bestimmte Änderungsarten wird die Änderungssignifikanz dem Sichtbarkeitsmodifikator angepasst. Zum Beispiel hat die Änderung des Rückgabe-Typs einer als *public* deklarierten Methode eine höhere Änderungssignifikanz als einer als *private* deklarierten Methode.

Mit Hilfe der AST-Informationen, können präzise Angaben zur Programmtext-Änderung gemacht werden. Eine Baum-Transformation kann zusätzlich zu den Informationen der geänderten Programmentität, erklären, wo die Änderung vorgenommen wurde. Zum Beispiel wird so präzise berechnet, dass der Methoden Aufruf `fred.bar()` in der Methode `calculate(int)` in die if-Anweisung mit der Bedingung `fred != null` verschoben wurde.

Um die Vielfältigkeit der von *Change Distilling* unterstützten Änderungsarten zu veranschaulichen, nehmen wir als Beispiel die folgenden zwei Versionen einer Methode:

Version 1:

```
public int foo(long num) {
   boolean check = num > 0;
   print("test");
   if (check) {
      return 42;
   } else {
      return 23;
   }
}
```

Version 2:

```
public int foo(int num) {
   boolean check = num > 0;
   if (!check) {
      return 42;
   } else {
      print("test");
      return 23;
   }
}
```

In der Deklaration der Methode `foo` wurde von Version 1 zu Version 2 der Typ des Parameters `num` von **`long`** auf **`int`** angepasst und hat die Änderungsart *parameter type change*. Die Änderungssignifikanz ist *kritisch*, da die Methode als *public* deklariert ist. Im Rumpf der Methode finden wir zwei Änderungen. Die erste verschiebt den Methodenaufruf `print("test");` in den else-Teil der if-Anweisung. Sie hat die Änderungsart *statement parent change*. Die zweite aktualisiert die Bedingung der if-Anweisung von `check` zu `!check` und hat die Änderungsart *control structure condition expression change*. Beide Änderungen erhalten die Änderungssignifikanz *mittel*, da sie zwar funktionlitätsändernd sein können, aber höchstwahrscheinlich keine Auswirkung auf die Nachbedingung der Methode haben.

ChangeDistiller. Der *Change-Distilling*-Algorithmus wurde als Werkzeug, *ChangeDistiller*, in die *Eclipse* IDE integriert. *ChangeDistiller* extrahiert feingranulare Programmtext-Änderungen aus der Entwicklungsgeschichte von Java Software Systemen und baut auf der *Evolizer* Platform auf [GFP09]. *Evolizer* stellt eine Persistenzschicht zur Verfügung, in der das Änderungsgeschichten- (*change history*) Meta-Modell integriert ist. So können einzelne Änderungen mit den Versionierungsinformationen eines Software Archivs verknüpft und in einer Datenbank gespeichert werden.

3 Gemeinsame Änderung von Kommentaren und Programmtext

Kommentare sind essentiell, speziell im Hinblick auf spätere Wartungsarbeiten. Sie ermöglichen den leichteren Einstieg für Personen, die noch nicht mit dem Software System vertraut sind. Den Entwicklern ist in der Regel die Wichtigkeit von Kommentaren bewusst, oft vernachlässigen sie aber die Wartung von bestehenden Kommentaren. Mögliche Gründe dafür sind Zeitdruck und Sorglosigkeit.

Eine quantitative Betrachtung der Verbindung zwischen Kommentaren und Programmtext lässt den Dokumentationsprozess von Programmtext besser verstehen. Die Analyse von gemeinsamen Änderungen zwischen Kommentaren und Programmtext gibt über eine gleichzeitige oder verzögerte Anpassung der Kommentare an den Programmtext Aufschluss. Wir nehmen dazu an, dass die meisten Kommentar-Änderungen aufgrund von Programmtext-Änderungen gemacht werden. Es ist zu erwarten, dass Kommentare selten in der gleichen Revision an Programmtext-Änderungen angepasst werden – Kommentare werden also retroaktiv angepasst.

Um die gemeinsamen Änderungen zwischen Kommentaren und Programmtext zu analysieren, wird ein Verfahren verwendet, dass zuerst Kommentare mit Programmentitäten verknüpft [FWG07] und danach mittels dem *Change-Distilling*-Algorithmus die gemeinsamen Änderungen extrahiert.

Da normalerweise ein Compiler eine Programmentität nicht automatisch zu den beschreibenden Kommentare bindet, sind Heuristiken von Nöten, um Verknüpfungen zwischen Kommentaren und einer Programmentitäten zu bestimmen. Dazu werden drei Heuristiken verwendet: (1) Ein Kommentar gehört zu einer Programmentität, wenn beide auf der glei-

chen Zeile sind; (2) Ein Kommentar zwischen zwei Programmentitäten gehört zur ersten Entität, wenn er durch Leerzeilen getrennt näher bei der ersten als bei der zweiten Entität ist. Ansonsten gehört er zur zweiten; und (3) Da ein Kommentar eine Programmentität beschreiben soll, lassen sich meistens Wörter aus dem Kommentar in der Entität finden. Bei dieser Heuristik wird der Deckungsgrad zwischen einem Kommentar und den möglichen Verknüpfungskandidaten bestimmt und so entschieden zu welchem Kandidaten der Kommentar gehört.

3.1 Empirische Studie der Kommentar-Änderungen

Die gemeinsamen Änderungen von Kommentaren und Programmtext wurden in sechs Software Systemen untersucht [FWGG09]. Zu diesen Systemen gehörten fünf öffentliche (*open-source*) und ein kommerzielles: Das UML-Zeichnungsprogramm *ArgoUML*, das Bittorrentprogramm *Azureus*, die Entwicklungsumgebung *Eclipse*, das Textbearbeitungsprogramm *jEdit*, die Diagrammapplikation *JFreeChart* und ein kommerzielles *Web-framework*.

In den Geschichten der untersuchten Systemen bleibt die relative Rate von Programmtext zu Kommentaren über die Zeit stabil. Sprich, das Wachstum von Kommentaren ist relativ gesehen gleich wie jenes vom Programmtext. Das bedeutet aber nicht zwingend, dass neuer Programmtext in ausreichendem Masse kommentiert wird, denn in der Hälfte der untersuchten Systemen wird weniger als 50% des Programmtext effektiv kommentiert.

Die Art der Programmentität beeinflusst, ob die Entität kommentiert wird oder nicht. Es gibt sogar eine gewisse Rangfolge in der Wahrscheinlichkeit, welche Art Entität kommentiert wird. Zum Beispiel werden Klassen öfter mit API-Kommentaren versehen, als Methoden oder Felder; Anweisungen wiederum weniger als Methoden; und einfache Anweisungen, wie z. B. einen Methodenaufruf, weniger als eine if-Anweisung.

Zu guter Letzt haben die effektiven Änderungen von Kommentaren und Programmtext gezeigt, dass gemeinsame Änderungen in 90% der Fälle in der gleichen Revision passieren. Ausnahmen sind API-Kommentare; diese werden zwar nicht in der gleichen Revision mit geändert, doch werden sie später nachkommentiert.

Schlussfolgerung. Die Resultate der Studie bestätigen teilweise, dass Programmtext wenig kommentiert ist. Der Anteil von kommentierten Code-Zeilen ist oft unter 50%. Obwohl die Entwickler der untersuchten Systeme sich der Wichtigkeit von Kommentare auf höherer Abstraktionsebene bewusst sind, werden trotzdem vor allem Methoden- und Klassen-Deklarationen zu wenig kommentiert.

Die gewonnenen Erkenntnisse helfen den Dokumentationsprozesse eines Software Systems zu beurteilen. Die gesammelten Änderungsdaten zeigen auf, wie Kommentar- und Programmtext-Zeilen wachsen. Damit lässt sich feststellen, ob Entwickler Programmtext generell kommentieren. Des Weiteren sehen wir, ob sich Entwickler der Wichtigkeit von Kommentaren bewusst sind. Falls dies nicht der Fall sein sollte, ist ein entsprechendes Training der Entwickler angebracht. Die Analyse der gemeinsamen Änderungen deckt die

Aktualität von Kommentaren auf und gibt an, ob der Dokumentationsprozess von Programmtext problematisch ist: Die Aktualität ist wichtig, um einen gewissen Grad an Verständlichkeit für neue Projekt-Mitglieder zu garantieren. Da nachträgliches Dokumentieren mehr Zeit benötigt als die sofortige Anpassung der Kommentare, kann das beschriebene Verfahren eine Entscheidungsgrundlage zur Entwicklungsprozess-Optimierung liefern.

Im folgenden Anwendungsszenario wird gezeigt, wie sich mittels Werkzeugunterstützung die dargelegten Probleme reduzieren lassen.

Anwendungsszenario. Nehmen wir an, der Software Entwickler Paul ändert sowohl die Deklaration einer Methode als auch deren Rumpf. Er passt den Typ eines Parameters der Methode an und entsprechend alle Referenzen in einer if-Anweisung. Die Methode hat einen API-Kommentar (z. B. Javadoc) und die if-Anweisung wird durch einen Kommentar unmittelbar vor der if-Anweisung beschrieben. Die Anweisungen innerhalb der if-Anweisung sind nicht kommentiert.

Nachdem Paul die beschriebenen Änderungen an der Methode vollzogen hat, wird ein Werkzeug innerhalb der Entwicklungsumgebung überprüfen, ob der API-Kommentar entsprechend der Deklarationsänderung angepasst wurde oder nicht. Ist dies nicht der Fall, schlägt das Werkzeug Paul vor, den API-Kommentar anzupassen. Zusätzlich wird das Werkzeug die Änderungsarten innerhalb der if-Anweisung sammeln und anhand deren Änderungssignifikanz entscheiden, ob eine Anpassung des Kommentars der if-Anweisung angebracht ist.

4 Gemeinsam auftretende Änderungsarten

Gewisse Programmtext-Änderungsarten werden meistens gemeinsam durch geführt. Zum Beispiel wirkt sich eine Umbenennung eines Parameters einer Methode auf alle Anweisung im Rumpf der Methode aus, die diesen Parameter referenzieren. Sprich, all diese Anweisungen müssen an die Umbenennung angepasst werden. So stellt sich die Frage, ob Muster von Änderungsarten existieren, welche über gewöhnliche *Refactorings* hinausgehen.

In [FGG08] wird dazu ein Verfahren vorgestellt, dass Muster von Änderungsarten mittels hierarchischem, agglomerativem Clustering findet. Clustering macht sich zu Nutzen, dass wiederholte Entwicklungsaktivitäten sich in bestimmten Gruppen von Änderungsarten widerspiegeln. Wenn z. B. zwei oder mehrere Änderungsarten im Laufe der Zeit immer wieder zusammen auftreten, dann lässt sich mit Hilfe des Clusterings ein Muster identifizieren.

4.1 Empirische Studie zu Muster von Änderungsarten

Das Clustering-Verfahren wurde sowohl auf die ganze als auch auf Quartalsabschnitte der Entwicklungsgeschichten von zwei öffentlichen und einem kommerziellen Software System angewendet: *jEdit*, *JFreeChart* und *Webframework*. Dazu wurde zwischen *globalen*

Mustern, d. h. Muster die über die ganze Geschichte verteilt immer wieder vorkommen, und *lokalen* Mustern, d. h. Muster die nur in bestimmten Zeitabschnitten vorkommen, unterschieden. Lokale Muster sollten Inkonsistenzen in den Änderungen hervor bringen; also Änderungen aufzeigen, die normalerweise in dieser Konstellation nicht angewendet werden. Als Beispiel nehmen wir an, *Exceptions* werden behandelt, indem sie zuerst gefangen und danach in einer benutzer-definierten *Exception* gekapselt weiter geworfen werden. Das entsprechende Muster der Änderungsarten ist dann {Einfügen eines *try*/*catch*-Blocks und einer *throw*-Anweisung}. Nun kam es vor, dass während einer gewissen Phase in der Entwicklungsgeschichte *Exceptions* anders behandelt werden, z. B. ohne die *throw*-Anweisung. Solche Inkonsistenzen können mit dem Clustering-Verfahren gefunden werden und somit Indikatoren für Fehler oder fehlende Programmierrichtlinien sein.

Die Studie an den drei Software Systemen brachten ein etwas anderes Bild hervor. Die gefundenen Muster von Änderungsarten zeigten auf, dass Ausnahmen für den normalen Programmfluss verwendet werden und dass die meisten Muster, die Änderungen im Kontrollfluss betreffen, mit dem „Aufräumen“ vom Programmtext zusammen hängen. Zum Beispiel wurden während einer gewissen Zeit im kommerziellen Software System alle *mutliple-exit* Methoden (d. h. mehrere Rückgabe-Anweisungen) zu *single-exit* Methoden (d. h. nur eine Rückgabe-Anweisung) umgebaut. Vereinheitlichungen von API-Konventionen sind hingegen über die ganze Entwicklungsgeschichte von Software Systemen verteilt. Ob die Notwendigkeit solcher Aufräumarbeiten durch inkonsistente Anwendung von Programmierrichtlinien oder durch häufiges Ändern der Richtlinien hervorgerufen werden, kann anhand diesen Resultaten nicht entschieden werden.

Schlussfolgerung. Mit der Erkennung von Mustern von Änderungsarten kann gezeigt werden, dass Programmierrichtlinien nicht konsequent von den Entwicklern befolgt werden, dass Projekt-Einsteiger nicht genügend geschult wurden oder dass Richtlinien oft ändern. Diese Erkenntnis ist wertvoll, da korrigieren unnötiger Änderungen am Programmtext aufwändig sind und verhindert werden können.

Im folgenden Anwendungsszenario wird gezeigt, wie sich mittels Werkzeugunterstützung die dargelegten Probleme reduzieren lassen.

Anwendungsszenario. Nehmen wir an, Paul stösst neu zu einem Entwicklungsteam. Das Team steht unter Zeitdruck und hat daher nur wenig Zeit Paul das Software System und die Programmierrichtlinien zu erklären – entsprechende Dokumentationen fehlen zur Zeit. Paul startet also sofort mit der ihm zugeteilten Aufgabe. Da sich Paul mit den Richtlinien noch nicht auskennt, hat er viele offene Fragen. Leider bringt ein Blick in bestehenden Programmtext keine Antworten. Nun gibt es zwei Möglichkeiten: Entweder Paul fragt jemanden erfahrenen oder er programmiert, wie er es sich gewohnt ist. Aufgrund des Zeitdrucks entscheidet Paul sich zu letzterem.

Mit dem Erkennen von Mustern von Änderungsarten kann eine dritte Möglichkeit realisiert werden. Ein in die Entwicklungsumgebung integriertes Werkzeug unterstützt Paul beim Programmieren, indem es vergangene Änderungsmuster lernt. Das Werkzeug kann entweder als Suchmaschine helfen, mit dem Paul gezielt nach Mustern sucht, oder es überprüft Paul's Änderungen und schlägt nötigenfalls Anpassungen vor.

5 Änderungen von Methodenaufrufen

Forschung mit dem Ziel, fehleranfällige Software-Module vorher zu sagen, kann auf eine lange Geschichte zurück blicken. Bereits in den siebziger Jahre wurden von McCabe [McC76] und Halstead [Hal77] Komplexitätsmetriken vorgeschlagen, um fehleranfällige Module zu ermitteln. Heutzutage sind fortgeschrittenere Methoden, welche historische Daten von Software Sytemen mit einbeziehen, vorhanden.

5.1 Fehlerbehebende Änderungen in Eclipse

Die Analyse von fehlerbehebenden, feingranularen Programmtext-Änderungen zeigt, ob es bestimmte fehlerbehebende Änderungsmuster gibt. Die Untersuchung solcher Änderungen in *Eclipse* brachte hervor, dass über 25% der Fehler mit weniger als vier Änderungen (Instanzen von Änderungsarten) behoben werden; z. B. zweimal Einfügen einer Anweisung (*statement inserts*) und eine Anpassung der Bedingung einer Kontroll-Struktur (*control structure condition expression change*). Zudem sind im Laufe der Entwicklungsgeschichte eine signifikante Anzahl Fehler mit dem Verschieben eines Methodenaufrufes in eine if-Anweisung (*statement parent change*), gelöst worden. Diese Beobachtungen legen die Annahme nahe, dass gewisse Methoden existieren, deren Aufrufe wesentlich stärker von solchen Änderungen betroffen sind als andere. Des Weiteren sollten Änderungen, die Aufrufe der gleichen Methode betreffen zu einander ähnlich sein und Änderungsmuster formen. Solche Muster zeigen auf, welche Methodenaufrufe speziell beachtet werden müssen, um frühzeitig potenzielle Fehler auszumerzen. Ein entsprechendes Werkzeug schlägt dann anhand der Muster Änderungen zur Entwicklungszeit vor. Ein entsprechendes Anwendungsszenario wird am Ende dieses Abschnittes vorgestellt.

Die Resultate von der *Eclipse* Studie waren viel versprechend. Interessanterweise werden Methodenaufrufe zur Standardbibliothek von Java am meisten geändert und sind in vielen Fehlerbehebungen involviert. Die gefundenen Änderungen formen Muster: Methodenaufrufe wurden immer wieder von if-Anweisungen umschlossen. Zum Beispiel wurde in *Eclipse* Methodenaufrufe zu `list.add(arg)` immer wieder in eine if-Anweisung mit der Bedingung `!list.contains(arg)` verschoben.

Schlussfolgerung. Muster fehlerbehebender Änderungen stärken das Bewusstsein für fehleranfällige Methodenaufrufe. Die Muster im Beispiel des `list.add(arg)`-Aufrufs unterstreichen, dass bei jedem neuen solchen Aufruf Vorsicht geboten ist, und dass ein Entwickler gegebenenfalls eine if-Anweisung einfügen sollte. Das Stärken dieses Bewusstseins kann natürlich dank Werkzeug-Unterstützung automatisiert werden. Ein entsprechender Prototyp, *ChangeCommander*, wird in [FZG08] vorgestellt.

Wie sich diese Werkzeugunterstützung einsetzen lässt, zeigt das folgende Anwendungsszenario.

Anwendungsszenario. Paul arbeitet mit der *Eclipse* IDE und ist damit beschäftigt neue Funktionalität zu implementieren. Dabei ändert er die Methode `resolveClassPath(..)`

in der Klasse `JavaProject` in der *Eclipse* Erweiterung `org.eclipse.jdt.core`.[2] In dieser Methode fügt Paul den Aufruf `resolvedEntries.add(rawEntry)` hinzu. Anhand der Aufruf-Information und den bereits extrahierten Muster aus der Vergangenheit von *Eclipse* kann *ChangeCommander* entsprechende Änderungsmuster finden und Paul vorschlagen. In diesem konkreten Fall können zwei Änderungen vorgeschlagen werden: (1) Füge eine if-Anweisung mit der Bedingung `resolvedEntries != null` ein und schiebe den Aufruf in die if-Anweisung; (2) Füge eine if-Anweisung mit der Bedingung `!resolvedEntries.contains(rawEntry)` ein und schiebe den Aufruf in die if-Anweisung. Falls Paul einen (oder beide) dieser Vorschläge auswählt, wird *ChangeCommander* die entsprechende Änderung automatisch vornehmen.

6 Résumé

Ein effektiver Weg, um die negativen Auswirkungen der kontinuierlichen Änderung eines Software Systems zu reduzieren, ist die Analyse von Programmtext-Änderungen. *Change Distilling* ist ein Ansatz, der die Software Evolutionsanalyse mit Programmtext-Änderungen bereichert. Mit der Definition und Klassifikation von Programmtext-Änderungsarten auf Basis von Baum-Transformationen im AST, können feingranulare Änderungen mittels dem *Change-Distilling*-Algorithmus extrahiert und klassifiziert werden.

Wie die Studien, die in diesem Artikel zusammen gefasst wurden, gezeigt haben, können sowohl interessante Erkenntnisse über die Entwicklungsgeschichte eines Software Systems gezogen werden, als auch, die extrahierten Informationen genutzt werden, um Entwickler in ihrer täglichen Arbeit zu unterstützen: (1) Die Analyse von gemeinsame Änderungen von Kommentaren und Programmtext beurteilt den Dokumentationsprozess von Programmtext quantitativ und unterstützt Entwickler beim Dokumentieren von Software. (2) Gruppen von oft miteinander auftretenden Änderungsarten heben hervor, ob Programmierrichtlinien konsequent angewandt werden oder ob sie immer wieder angepasst werden. Ein Werkzeug, das Muster aus diesen Gruppen lernt, unterstützt Entwickler als Richtlinien-Suchmaschine oder gibt Vorschläge, falls die Richtlinien nicht konsistent angewendet werden. (3) Die Analyse von fehlerbehebenden Änderungsmuster stärkt das Bewusstsein für fehleranfällige Methodenaufrufe. Ein entsprechendes Werkzeug macht sich diese Muster zu Nutze und gibt Entwicklern Änderungsvorschläge während dem Programmieren.

Danksagungen. Ich bedanke mich herzlich bei meinem Doktorvater, Harald C. Gall, für seine stets ermutigende Unterstützung während meines Studiums und für sein Engagement mir optimale Arbeitsbedingungen zum Forschen zu schaffen. Weiter bedanke ich mich bei Michael Würsch und Gerald Reif für die wertvollen Kommentare zu diesem Artikel.

[2] Das Szenario ist zwar fiktiv, die Erweiterung mit der besprochenen Klasse und Methode gibt es hingegen wirklich.

Literatur

[FG06] Beat Fluri und Harald C. Gall. Classifying Change Types for Qualifying Change Couplings. In *Proc. Int'l Conf. Program Comprehension*, Seiten 35–45, 2006.

[FGG08] Beat Fluri, Emanuel Giger und Harald C. Gall. Discovering Patterns of Change Types. In *Proc. Int'l Conf. Automated Software Eng.*, Seite 4, 2008.

[Flu08] Beat Fluri. *Change Distilling – Enriching Software Evolution Analysis with Fine-Grained Source Code Change Histories*. Dissertation, University of Zurich, 2008.

[FWG07] Beat Fluri, Michael Würsch und Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In *Proc. Working Conf. Reverse Engineering*, Seiten 70–79, 2007.

[FWGG09] Beat Fluri, Michael Würsch, Emanuel Giger und Harald C. Gall. Analyzing the Co-Evolution of Comments and Source Code. *Software Quality Journal*, 2009.

[FWPG07] Beat Fluri, Michael Würsch, Martin Pinzger und Harald C. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[FZG08] Beat Fluri, Jonas Zuberbühler und Harald C. Gall. Recommending Method Invocation Context Changes. In *Proc. Int'l Workshop Recommender Systems for Software Eng.*, Seiten 1–5, 2008.

[GFP09] Harald C. Gall, Beat Fluri und Martin Pinzger. Change Analysis with Evolizer and ChangeDistiller. *IEEE Software*, 26(1):26–33, 2009.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier Science, Inc., 1977.

[Leh80] Many M. Lehman. Programs, Life Cycles and Laws of Software Evolution. *Proc. IEEE*, 68(9):1060–1076, 1980.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

Beat Fluri wurde am 16.07.1979 in Basel geboren. Nach Erreichen der Maturität am Gymnasium Kirschgarten in Basel begann er 1999 sein Informatik-Studium an der Eidgenössischen Technischen Hochschule in Zürich, welches er 2004 als MSc ETH in Computer Science abschloss. Danach war er von 2004 bis 2008 Forschungsassistent und Doktorand der Universität Zürich am Lehrstuhl für Software Engineering von Prof. Dr. Harald C. Gall. Dort promovierte Beat Fluri im Oktober 2008 mit dem Prädikat summa cum laude und erhielt von der Wirtschaftswissenschaftlichen Fakultät den Jahrespreis der Universität Zürich für seine Dissertation. Zur Zeit arbeitet er als Senior Research Associate bei Prof. Dr. Harald C. Gall.